Architecture Design of Area-Efficient SRAM-Based Multi-Symbol Arithmetic Encoder in H.264/AVC

Yu-Jen Chen, Chen-Han Tsai, and Liang-Gee Chen
Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan
{yjchen, chtsai, lgchen}@video.ee.ntu.edu.tw

Abstract— **The first SRAM-based multi-symbol arithmetic encoder was proposed in this paper. Since several SRAM problems arise in this highly data-dependent operation, four methods were introduced to make it feasible. Based on data-forwarding architecture, modular banks with throw-backward/catch-forward and read/write isolation greatly enhanced the throughput. Our SRAM-based approach was implemented with 29%–35% of area compared to register-based design. Moreover, different throughput required in various applications could be attained by changing the number of SRAM banks. The proposed SRAM-based multi-symbol arithmetic encoder achieved high throughput and low cost at the same time.**

I. Introduction

H.264 [1] is a state-of-the-art video coding standard. It attains high quality with relatively low bit-rate. Context-Based Adaptive Binary Arithmetic Coding (CABAC), the entropy coding adopted in main and high profiles, contributes significant bit-rate savings. Fig. 1 shows its block diagram. Syntax elements (*SE*) are the data to be coded. Side information is mostly the knowledge of neighboring coded blocks. *SE*s are transformed into binary *symbol*s for binary arithmetic coding. To achieve adaptive effect, *symbol*s are classified into many categories, i.e. contexts (*ctx*), which are modelled based on *SE* type, side information and bin index. *Symbol*s with the same *ctx* have similar statistic property and share the adaptive probability state. Besides normal arithmetic coding, bypass mode is introduced to speed up encoding process. Then, binary *symbol*s along with associated *ctx*s and *bypass* flags are passed to arithmetic encoder (AE). Finally, *bitstream* is generated.

Arithmetic coding is a recursive subdivision scheme of an interval, which is specified by *range* and *low*. The binary *symbol* is regarded as either the Most Probable Symbol (*MPS*) or the Least Probable Symbol (*LPS*). First, the probability state, composed of {*state, MPS*}, is acquired according to the *ctx* of *symbol*. Depending on the *symbol* equals *MPS* or not, next interval is updated as one of two sub-intervals, as shown in Fig. 2, where *rangeLPS* depends on *state* and *range*.

In this paper, an area-efficient SRAM-based multi-symbol AE is proposed. The motivation and challenges of this work are discussed in Section II and III. The architecture is developed in Section IV. Section V shows implementation results and comparisons. Section VI is the conclusion.

II. Motivation of SRAM-Based AE

CABAC in H.264 uses 460 *ctx*s to attain accurate probability estimation, and each *ctx* stores one {*state, MPS*} pair in memory. Compared with 17 *ctx*s in JPEG2000, 460 *ctx*s in H.264 is incredibly large. The huge amount of {*state, MPS*} memory is a whole new problem first arising in H.264. Furthermore, for high-end applications, e.g. HDTV, high-throughput AE increases the hardware costs greatly. In our previous work [2], the architecture of multi-symbol AE successfully achieves high throughput, but its huge area is still a problem.

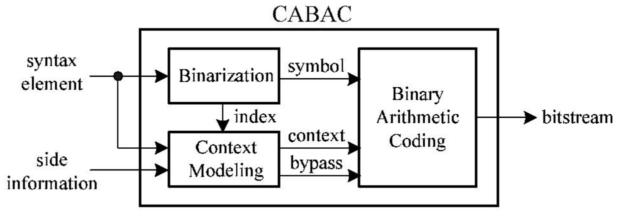

Fig. 1. Block diagram of H.264/AVC CABAC

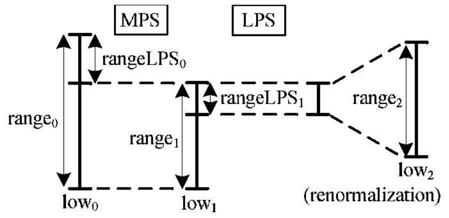

Fig. 2. Conceptual arithmetic coding in H.264/AVC

To find an area-efficient solution, the area profiling of the architecture in [2] is analyzed first, as shown in Fig. 3. *State stage*s occupies about 95% of area in one-, two-, and four-symbol conditions. As expected, the huge {*state, MPS*} memory is the area-dominant part. The major work of *state stage* is to read {*state, MPS*} from memory, update {*state, MPS*}, and then write back to memory. In [2], the {*state, MPS*} memory is implemented by use of registers, which has several drawbacks. Firstly, lots of scattered registers and complex wires enlarge the area considerably. Secondly, register-based architecture introduces additional 460 comparators and multiplexers. Thirdly, all the 460 {*state, MPS*} data can be accessed in one cycle, but only a few of them (equals the number of encoded symbols per cycle) are in use. This is quite a waste. Therefore, register-based architecture is unsuitable for AE application. On the contrary, SRAM is much more compact than registers, about 30% of area only. Thus, we choose SRAM as {*state, MPS*} memory in this work.

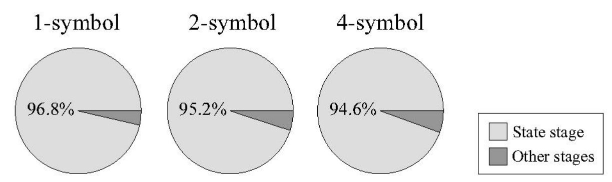

Fig. 3. Area profiling

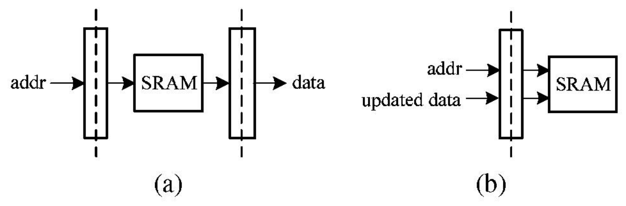

Fig. 4. Block diagram of (a) SRAM read (b) SRAM write

III. Challenges of SRAM-Based AE

SRAM is quite efficient in area, but it has two fatal problems when applied in AE, a highly data-dependent process. Two main challenges of SRAM-based AE are described as follows.

A. Bubbles

In general, registers are inserted at both input and output of SRAM read/write to assure enough drive strength. This is shown in Fig. 4, where the SRAM read/write is negative-edge-triggered. Due to the use of SRAM, the work in *state stage* takes four cycles, i.e. Address Generation (*AG*), *SRAM read*, *update* and *SRAM write*. Fig. 5 shows the data-dependent bubbles in *state stage*. Before $symbol_1$ finishes *SRAM write*, the incoming *symbol*s with the same *ctx* can not read the up-to-date data from SRAM. For this reason, it needs to idle two cycles after working one cycle. In contrast with register-based AE, the two bubbles degrade the throughput to 1/3.

B. Insufficient ports

In s-symbol AE, s data are read from SRAM, and another s updated data are written to SRAM in the same cycle, so in total $2s$ ports is required. But in general, at most two-port SRAM is available. Two ports are sufficient for one-symbol AE only. Limited to the number of ports, high processing rate of multi-symbol AE is cancelled by the throughput degradation.

Some literature of H.264 AE does not implement or partially implements *state stage* [3][4]. Previously mentioned multi-symbol AE [2] includes complete *state stage*, but suffers from huge area of {*state, MPS*} registers. In [5], SRAM-based one-symbol AE is proposed. However, it encodes at most one symbol per cycle under the limitation of SRAM ports. Besides, because of SRAM bubbles, its throughput in average drops to 1/3 symbol per cycle, which is quite insufficient in high-end applications. Up to now, AE with both high throughput and low cost does not show up. Therefore, to meet the needs of various applications, this paper is devoted to the design of an area-efficient SRAM-based multi-symbol AE.

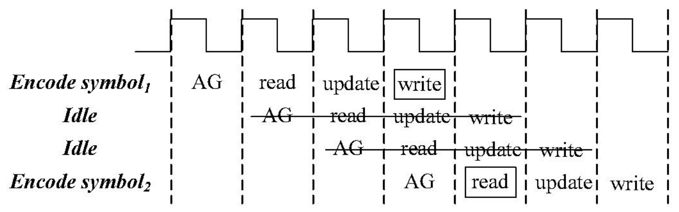

Fig. 5. Bubbles in SRAM-based AE

IV. Proposed Architecture

In the design of SRAM-based multi-symbol AE, the key point is to access the up-to-date {*state, MPS*} in time and also attain high throughput. This section focuses on this key point only. Other parts of AE adopt multi-symbol architecture in [2].

A. SRAM-Based One-Symbol AE

The only problem in one-symbol case is the two bubbles between working cycles (Fig. 5). To remove the two bubbles, the latest {*state, MPS*} shall be available in the two idle cycles even if these *ctx*s are the same. The proposed solution is data-forwarding, i.e. to store data for the use in the near future. The architecture is shown in Fig. 6. There are two paths to access {*state, MPS*}. One is SRAM path, and the other is register path. Normally, if the *ctx* of current *symbol* is different from its prior two *ctx*s, it gets {*state, MPS*} from SRAM path successfully. Otherwise, data-forwarding is performed. The up-to-date {*state, MPS*} is fed into register path in either *AG* or *read stage*, and then the data in register path is selected in *update stage* (Fig. 6). When the type of a *symbol* is *bypass* or *termination* (*ctx*=276), register path is also selected because its {*state, MPS*} is constant and unnecessary to be accessed from SRAM. *Data-forwarding architecture* guarantees that whatever the *ctx*s are, all *symbol*s can get correct {*state, MPS*} without idle cycles. Therefore, the throughput of 1/3 symbol per cycle is greatly improved to one symbol per cycle with very little area overhead, roughly two {*state, MPS*} registers, two *ctx* comparators, and three multiplexers.

The architecture in Fig. 6 can be simplified to the symbolic representation in Fig. 7 (a). A, B, C, and D in Fig. 7 (a) correspond to those in Fig. 6. In *AG stage*, *ctx* of A is compared to its prior two *ctx*s (*ctx*s of B and C). Depending on this result, when it is shifted to *update stage* two cycles later, C is either read from SRAM (SRAM path) or shifted from B (register path). C is taken to look up *rangeLPS* for remaining AE procedure. If *ctx* of C equals *ctx* of A or B, D (the update of C) is passed to A or B. To avoid too complicated figures, the following discussion on multi-symbol AE is based on this symbolic representation.

B. SRAM-Based Multi-Symbol AE

Besides data-dependent bubbles, the problem of insufficient ports arises in multi-symbol AE. Thus the throughput in every cycle is not constant any more. For s-symbol AE, the encoded symbols per cycle range between one and s, so the

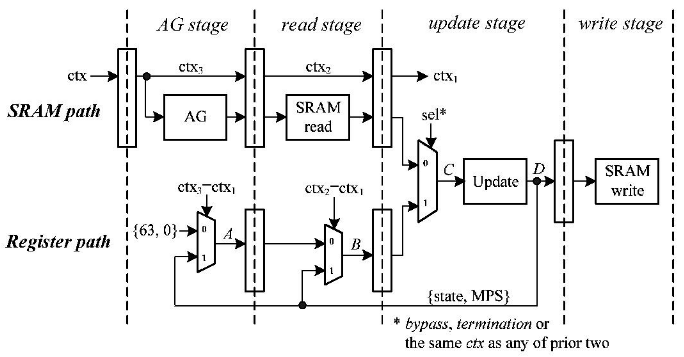

Fig. 6. *Data-forwarding architecture* in SRAM-based one-symbol AE

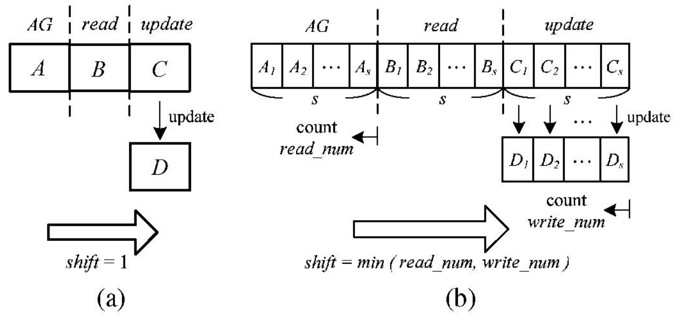

Fig. 7. Symbolic representation of (a) one-symbol (b) multi-symbol AE

average throughput is far below s. Based on *data-forwarding architecture*, several methods are proposed to enhance the throughput.

1) Data-forwarding architecture: Fig. 7 (b) is the symbolic representation of SRAM-based s-symbol AE. In *AG stage*, *ctx*s of A_1–A_s are compared to those prior to them. For example, A_2 should be compared to *ctx*s of A_3–A_s, B_1–B_s, and C_1–C_s. If the *ctx* differs from all preceding *ctx*s, it gets {*state, MPS*} from SRAM path two cycles later. Otherwise, it is ensured getting {*state, MPS*} from register path sooner or later because D_1–D_s are passed backward before leaving *update stage* if any same *ctx*s follow them. When a *ctx* equals more than one prior *ctx*, it accepts {*state, MPS*} from the nearest (latest) one. Again, *data-forwarding architecture* removes two idle cycles in multi-symbol AE.

2) Modular banks: The remaining problem is insufficient SRAM ports. How SRAM ports affects the throughput is explained as follows. In Fig. 7 (b), *symbol*s of A_1–A_s/D_1–D_s are defined as *readable/writable* if they can access SRAM read/write ports or exploit register path. Of course, the prior *symbol*s have higher priority to access SRAM ports. *Read_num* in Fig. 7 (b) is the number of consecutive *readable symbol*s counted from A_s to left; *write_num* is the number of consecutive *writable symbol*s counted from D_s to left. The throughput in current cycle, *shift* in Fig. 7 (b), is the minimum of *read_num* and *write_num* to guarantee that {*state, MPS*} is always available before or in *update stage*, and updated {*state, MPS*} is always ready to be written to SRAM before leaving *update stage*. Except register path, most *symbol*s have to read from and write to SRAM. Therefore, the problem of insufficient SRAM ports degrades the throughput seriously.

The direct approach to increase SRAM ports is to partition 460 *ctx*s into several SRAM banks. If two-port SRAM is used, b banks have $2b$ ports, which seems sufficient for b-symbol AE. However, since each bank has only one read port and one write port, *collision* occurs when more than one *ctx* has to read from or write to the same bank concurrently. Frequent *collision*s reduces *readable* and *writable symbol*s, leading to great throughput loss. Therefore, it is very important to find an adequate partitioning method to lower the probability of *collisions*. In addition, regular methods are preferred in hardware implementation. To meet the two requirements, the formation of *ctx*s shall be investigated first. *Ctx* is the sum of *ctxIdxOffset*, *ctxIdxBlockCatOffset*, and *ctxIdxInc* [1]. *Ctx*s of neighboring *symbol*s usually differ from each other on *ctxIdxInc*s only, and the latter *ctxIdxInc* is often greater than the former by one. From this observation, *ctx*s of neighboring *symbol*s tend to scatter in different *modular banks*, i.e. every *ctx* is assigned to *(ctx % b)*-th bank if b banks are adopted. Due to the regularity and fitness for *ctx* formation, we choose *modular banks* to partition *ctx*s.

3) Throw-backward/catch-forward: Besides removing idle cycles, *data-forwarding architecture* brings another beneficial effect. When two *ctx*s are the same, the former throws updated {*state, MPS*} backward, and the latter catches it. Since the former {*state, MPS*} is thrown backward, it is unnecessarily written to SRAM, which can save the use of write port; since the latter *ctx* catches forward {*state, MPS*}, it unnecessarily read data from SRAM through a read port. There is great probability of the same *ctx*s between neighboring *symbol*s in H.264. Dealing with this situation is originally troublesome, but now it becomes an advantage of suppressing port demand, and thus reducing the occurrence of *collision*s.

4) Read/write isolation: The throughput (*shift*) is the minimum of *read_num* and *write_num*. Unfortunately, they may be very different in a cycle, which degrades the throughput drastically. The solution is to isolate read/write operation in *AG/update stage* respectively, regardless of the condition of the other side. Whatever *shift* is, we read and write as many *symbol*s as possible, some of which are prepared for future cycles. For example, in four-symbol AE, if A_4 and A_1 are *readable* (*read_num*=1), and D_4, D_3, and D_1 are *writable* (*write_num*=2), the throughput in this cycle is only one. Obviously, all three data can be written to SRAM or backward registers safely. Besides A_4, we can also read A_1 from SRAM in advance even if any preceding *ctx* is equal to the *ctx* of A_1, because it is possibly overridden by the correct value before or in *update stage*. As a result, *read/write isolation* tends to increase the throughput in future cycles.

V. Implementation Results

In Section IV, four methods are proposed to overcome challenges of SRAM and improve the throughput. *Data-forwarding architecture* removes data-dependent bubbles.

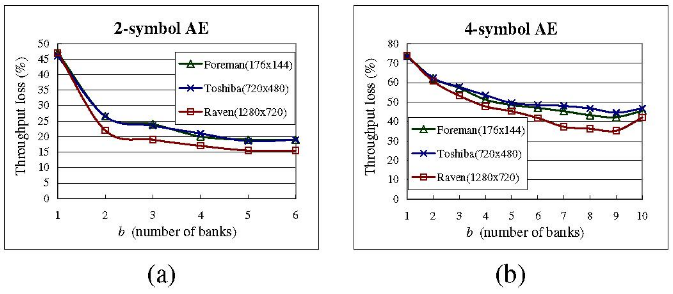

Fig. 8. Performance of *modular banks* only in (a) 2-symbol (b) 4-symbol AE

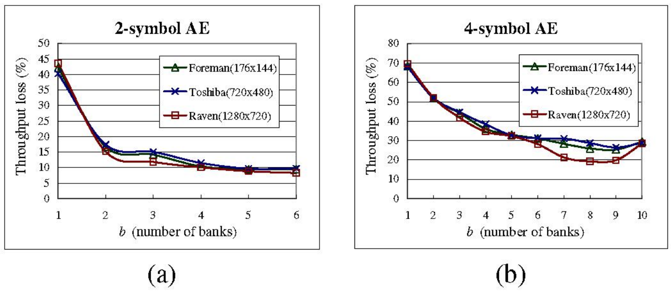

Fig. 9. Performance of *modular banks* with *throw-backward/catch-forward* in (a) 2-symbol (b) 4-symbol AE

Modular banks increase the number of ports and fit *ctx* formation in H.264. *Throw-backward/catch-forward* and *read/write isolation* move more data to register path, so reduce the use of ports and enhance the throughput in current and future cycles. With these techniques, SRAM-based multi-symbol AE can be realized with high precessing rate.

We simulated several sequences from QCIF to HDTV, with low to high motion. The effects of these four schemes are presented as follows. *Data-forwarding architecture* enhances the throughput up to three times. The performance of *modular banks* is shown in Fig. 8, in which the throughput loss is degradation percent of maximum *symbols* encoded in a cycle. Note that the improvement does not always increase with the number of banks, e.g. the degradation of 10 banks in four-symbol AE. This is because *ctx* arrangement between different *SE*s is unsuitable for certain modulus. After *throw-backward/catch-forward* applied, the results are shown in Fig. 9. Finally, Fig. 10 shows the performance with the help of both *throw-backward/catch-forward* and *read/write isolation*. In two-symbol AE with two banks, the throughput loss is reduced from 25%, 16% to 8.2%; in four-symbol AE with four banks, the throughput loss is reduced from 51%, 36% to 17%. The great improvement is quite stable among various sequences.

The proposed SRAM-based multi-symbol AE is implemented in TSMC 0.18 μm technology. Table I shows the comparisons with previous works. This is the first SRAM-based multi-symbol AE, so only SRAM-based one-symbol AE [5] and register-based multi-symbol AE [2] are compared. In Table I, Ours-2 is two-symbol AE with two banks, and Ours-4

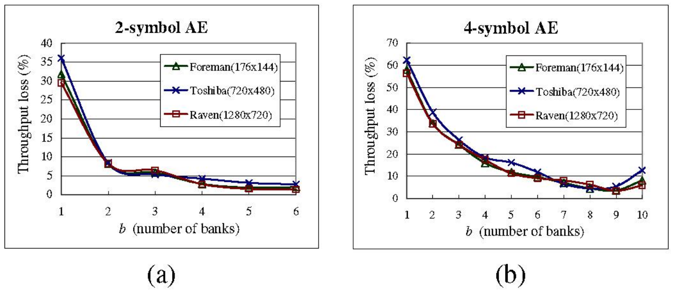

Fig. 10. Performance of *modular banks* with *throw-backward/catch-forward* and *read/write isolation* in (a) 2-symbol (b) 4-symbol AE

TABLE I
COMPARISONS OF DIFFERENT WORKS (ALL IMPLEMENTED IN 0.18 μM TECHNOLOGY)

	Memory type	Symbols/cycle	Gate count	Critical path
[5]-1	SRAM	1/3	-	3.8ns
[2]-1	Register	1	45.8K	2.4ns
[2]-2	Register	2	61.4K	3.1ns
[2]-4	Register	4	92.9K	5.2ns
Ours-1	SRAM	1	13.2K	1.6ns
Ours-2	SRAM	1.84	18.9K	2.9ns
Ours-4	SRAM	3.32	32.1K	5.2ns

is four-symbol AE with four banks. The throughput different from register-based AE is 8.2% and 17%. If higher throughput is desired, the number of *modular banks* can be increased. The area of SRAM-based one-, two-, and four-symbol AE is only 29%, 31%, and 35% compared to [2]. Moreover, the critical paths of one- and two-symbol AE are shortened because in [2]-1 and [2]-2, the critical paths fall in register-based *state stages*, which are longer than SRAM-based *state stages*.

VI. Conclusion

In this paper, the first SRAM-based multi-symbol AE is implemented. AE in H.264 suffers from huge area of memory due to lots of *ctx*s, so SRAM is used instead of registers. Several methods are developed to solve SRAM bubbles and insufficient ports. Finally, we propose an area-efficient SRAM-based multi-symbol AE with less than 35% of area compared to previous work in one-, two- and four-symbol AE. Also, different throughput can be achieved by changing the number of SRAM banks. Besides H.264, the architecture can be applied to other standards as well.

References

[1] Joint Video Team, *Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification*, ITU-T Rec. H.264 and ISO/IEC 14496-10 AVC, May 2003.

[2] C. H. Tsai, Y. J. Chen, and L. G. Chen, "Analysis and architecture design for multi-symbol arithmetic encoder in h.264/avc," in *Proc. of ISOCC*, 2005.

[3] R.-R. Osorio and J.-D. Bruguera, "Arithmetic coding architecture for h.264/avc cabac compression system," in *Proc. of DSD*, 2004.

[4] J. L. Nunez and V. A. Chouliaras, "High-performance arithmetic coding vlsi macro for the h.264 video compression standard," *IEEE Transactions on Consumer Electronics*, vol. 51, no. 1, pp. 144–152, Feb. 2005.

[5] H. Shojania and S. Sudharsanan, "A high performance cabac encoder," in *Proc. of NEWCAS*, 2005.